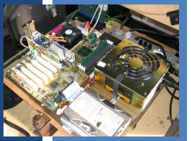
TCSS 422: OPERATING SYSTEMS



**Three Easy Pieces:
Lock Based Data Structures,
Condition Variables**

Wes J. Lloyd
Institute of Technology
University of Washington - Tacoma

OBJECTIVES

- Assignment 1 – MASH Shell
- Lock Based Data Structures – Ch. 29
- Condition Variables – Ch. 30
- Quiz 3 – Lock-Based Data Structure Coding Activity

FEEDBACK – 4/23

- Do we have to know all the variations of Spin Locks?
  - Ch. 28 is just too much
- Basic spin lock
  - Polling / busy-waiting
  - When is it appriorate?
- Test-and-set spin lock
- Compare-and-swap spin lock
  - ***It is good to know what each successive version adds***
- Can you give us more practice questions on calculating the average response and turnaround time
  - Practice midterm next Monday 4/30

FEEDBACK – HW1

- We don't understand:
  "While writing your MASH program using processes, consider why it is non-trivial to simply redirect the output stream of each fork command to a stream and use the C sscanf() function to consolidate/aggregate the output at the end… "
- ***Comment is ambiguous***
- ***In C (and Java) you could open arbitrary input and output streams that are not associated with a file on the disk. It would then be possible to redirect each exec command's output stream to streams not associated with files.***
- ***Idea is to have "temporary in-memory buffers", in place of sending output to temporary files on the disk.***
- ***I did not try this.***
- ***For this assignment, it is easy enough to follow the example code and redirect exec output to temporary files:***

  http://faculty.washington.edu/wlloyd/courses/tcss422/examples/exec2.c

HW1 - 2

- We are executing commands fine and writing output. However, when it comes to the final process of displaying the output files in the correct order, it seems intuitive to use the Linux system() command.
- Is this acceptable?
- ***This solution works, but it shouldn't be difficult to write a C routine that opens a file, reads it line-by-line, and displays output.***
- ***This could be a generic, standalone routine.***
- ***Example C should exist online by searching via Google to support accomplishing this.***

HW1 - 3

- For notification of process completion, the assignment states:
  First process finished…
  Second process finished…
  Third process finished…
- Should it always be that order? Or could it be:
  Third process finished…
  First process finished…
  Second process finished…
- ***Reporting the order in which specific processes end is not required. Just report that processes *ARE* ending !***
- ***There is no requirement to say which child process/PID finishes in what order, etc.***
- ***FEATURE: Provides a notification message stating processes are finishing and work is proceeding.***

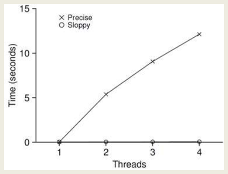
CHAPTER 29 – LOCK BASED DATA STRUCTURES

LOCK-BASED CONCURRENT DATA STRUCTURES

- Adding locks to data structures make them thread safe.
- Considerations:
  - Correctness
  - Performance
  - Lock granularity

CONCURRENT COUNTERS - PERFORMANCE

- iMac: four core Intel 2.7 GHz i5 CPU
- Each thread increments counter 1,000,000 times



Traditional vs. sloppy counter
Sloppy Threshold (S) = 1024

Synchronized counter scales poorly.

PERFECT SCALING

- Achieve (N) performance gain with (N) additional resources
- Throughput:
- Transactions per second
- 1 thread/core
- N = 100 tps
- 10 threads/cores
- N = 1000 tps

SLOPPY COUNTER

- Provides single logical shared counter
  - Implemented using local counters for each ~CPU core
    - 4 CPU cores = 4 local counters & 1 global counter
    - Local counters are synchronized via local locks
  - Global counter is updated periodically
    - Global counter has lock to protect global counter value
    - Sloppiness threshold (S): Update threshold of global counter with local values
    - Small (S): more updates, more overhead
    - Large (S): fewer updates, more performant, less synchronized
- Why this implementation? Why do we want counters local to each CPU Core?

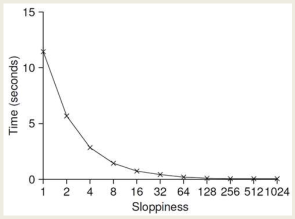
SLOPPY COUNTER - 2

- Update threshold (S) = 5
- Synchronized across four CPU cores
- Threads update local CPU counters

Time	L_1	L_2	L_3	L_4	G
0	0	0	0	0	0
1	0	0	1	1	0
2	1	0	2	1	0
3	2	0	3	1	0
4	3	0	3	2	0
5	4	1	3	3	0
6	5 → 0	1	3	4	5 (from L_1)
7	0	2	4	5 → 0	10 (from L_4)

THRESHOLD VALUE *S*

- Consider 4 threads increment a counter 1000000 times each
- Low *S* → What is the consequence?
- High *S* → What is the consequence?



SLOPPY COUNTER - EXAMPLE

- Example implementation
- Also with CPU affinity

CONCURRENT LINKED LIST - 1

- Simplification - only basic list operations shown
- Structs and initialization:

```
        // basic node structure
        typedef struct __node_t {
                int key;
                struct __node_t *next;
        } node_t;

        // basic list structure (one used per list)
        typedef struct __list_t {
                node_t *head;
                pthread_mutex_t lock;
        } list_t;

        void List_Init(list_t *L) {
                L->head = NULL;
                pthread_mutex_init(&L->lock, NULL);
        }

(Cont.)
```

CONCURRENT LINKED LIST - 2

- Insert – adds item to list
- Everything is critical!
  - There are two unlocks

```
(Cont.)
        int List_Insert(list_t *L, int key) {
                pthread_mutex_lock(&L->lock);
                node_t *new = malloc(sizeof(node_t));
                if (new == NULL) {
                        perror("malloc");
                        pthread_mutex_unlock(&L->lock);
                return -1; // fail
                new->key = key;
                new->next = L->head;
                L->head = new;
                pthread_mutex_unlock(&L->lock);
                return 0; // success
        }
(Cont.)
```

CONCURRENT LINKED LIST - 3

- Lookup – checks list for existence of item with key
- Once again everything is critical
  - Note - there are also two unlocks

```
(Cont.)

        int List_Lookup(list_t *L, int key) {
                pthread_mutex_lock(&L->lock);
                node_t *curr = L->head;
                while (curr) {
                        if (curr->key == key) {
                                pthread_mutex_unlock(&L->lock);
                                return 0; // success
                        }
                        curr = curr->next;
                }
                pthread_mutex_unlock(&L->lock);
                return -1; // failure
        }
```

CONCURRENT LINKED LIST

- First Implementation:
  - Lock **everything** inside Insert() and Lookup()
  - If malloc() fails lock must be released
    - Research has shown "***exception-based control flow***" to be error prone
    - 40% of Linux OS bugs occur in rarely taken code paths
    - Unlocking in an exception handler is considered a poor coding practice
    - There is nothing specifically wrong with this example however
- Second Implementation …

CCL – SECOND IMPLEMENTATION

- Init and Insert

```
void List_Init(list_t *L) {
        L->head = NULL;
        pthread_mutex_init(&L->lock, NULL);
}

void List_Insert(list_t *L, int key) {
        // synchronization not needed
        node_t *new = malloc(sizeof(node_t));
        if (new == NULL) {
                perror("malloc");
                return;
        }
        new->key = key;

        // just lock critical section
        pthread_mutex_lock(&L->lock);
        new->next = L->head;
        L->head = new;
        pthread_mutex_unlock(&L->lock);
}

```

CCL – SECOND IMPLEMENTATION - 2

- Lookup

```
(Cont.)
int List_Lookup(list_t *L, int key) {
        int rv = -1;
        pthread_mutex_lock(&L->lock);
        node_t *curr = L->head;
        while (curr) {
                if (curr->key == key) {
                        rv = 0;
                        break;
                }
                curr = curr->next;
        }
        pthread_mutex_unlock(&L->lock);
        return rv; // now both success and failure
}
```

CONCURRENT LINKED LIST PERFORMANCE

- Using a single lock for entire list is not very performant
- Users must "wait" in line for a single lock to access/modify any item
- Hand-over-hand-locking (lock coupling)
  - Introduce a lock for each node of a list
  - Traversal involves handing over previous node's lock, acquiring the next node's lock…
  - Improves lock granularity
  - Degrades traversal performance
- Consider hybrid approach
  - Fewer locks, but more than 1
  - Best lock-to-node distribution?

MICHAEL AND SCOTT CONCURRENT QUEUES

- Improvement beyond a single master lock for a queue (FIFO)
- Two locks:
  - One for the **head** of the queue
  - One for the **tail**
- Synchronize enqueue and dequeue operations
- Add a dummy node
  - Allocated in the queue initialization routine
  - Supports separation of head and tail operations
- Items can be added and removed by separate threads at the same time

CONCURRENT QUEUE

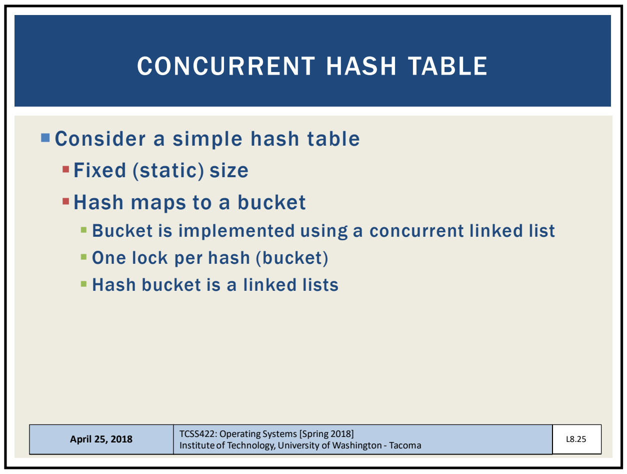
- Remove from queue

```
typedef struct __node_t {
        int value;
        struct __node_t *next;
} node_t;

typedef struct __queue_t {
        node_t *head;
        node_t *tail;
        pthread_mutex_t headLock;
        pthread_mutex_t tailLock;
} queue_t;

void Queue_Init(queue_t *q) {
        node_t *tmp = malloc(sizeof(node_t));
        tmp->next = NULL;
        q->head = q->tail = tmp;
        pthread_mutex_init(&q->headLock, NULL);
        pthread_mutex_init(&q->tailLock, NULL);
}

(Cont.)
```

CONCURRENT QUEUE - 2

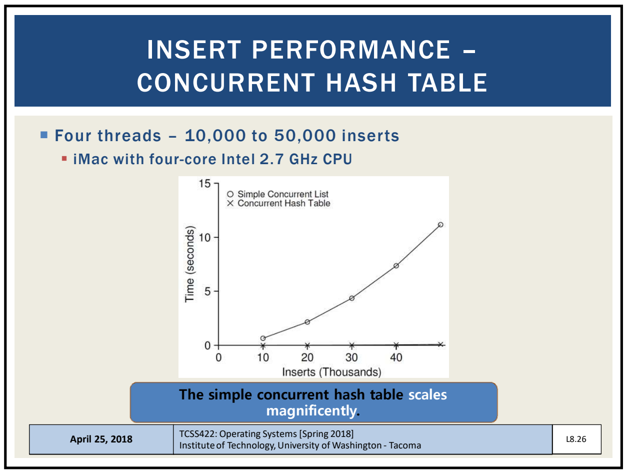
- Add to queue

```
(Cont.)
void Queue_Enqueue(queue_t *q, int value) {
        node_t *tmp = malloc(sizeof(node_t));
        assert(tmp != NULL);

        tmp->value = value;
        tmp->next = NULL;

        pthread_mutex_lock(&q->tailLock);
        q->tail->next = tmp;
        q->tail = tmp;
        pthread_mutex_unlock(&q->tailLock);
}
(Cont.)
```



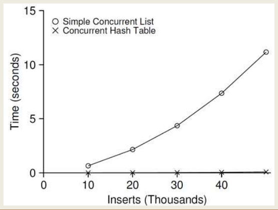
CONCURRENT HASH TABLE

- Consider a simple hash table
  - Fixed (static) size
  - Hash maps to a bucket
    - Bucket is implemented using a concurrent linked list
    - One lock per hash (bucket)
    - Hash bucket is a linked lists

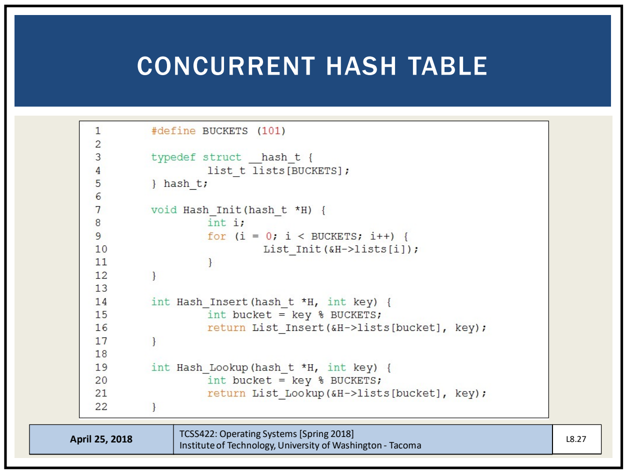


INSERT PERFORMANCE – CONCURRENT HASH TABLE

- Four threads – 10,000 to 50,000 inserts
  - iMac with four-core Intel 2.7 GHz CPU

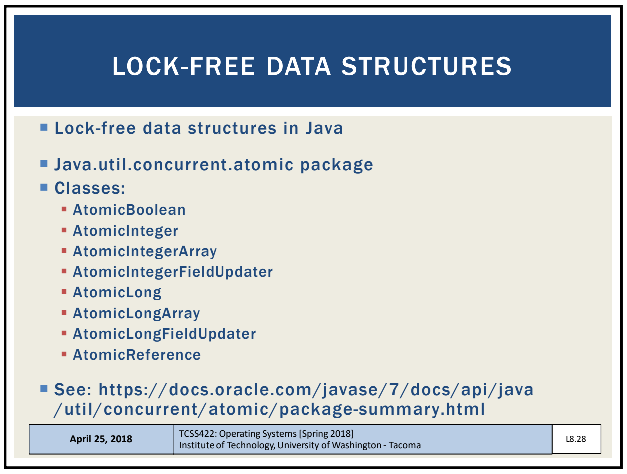


The simple concurrent hash table scales magnificently.



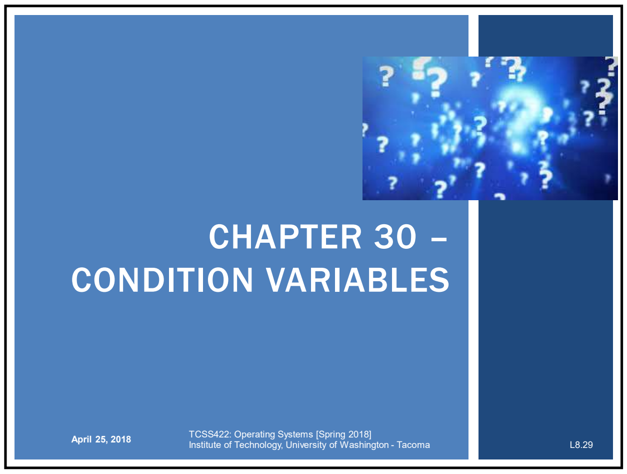
CONCURRENT HASH TABLE

```
#define BUCKETS (101)

typedef struct __hash_t {
        list_t lists[BUCKETS];
} hash_t;

void Hash_Init(hash_t *H) {
        int i;
        for (i = 0; i < BUCKETS; i++) {
                List_Init(&H->lists[i]);
        }
}

int Hash_Insert(hash_t *H, int key) {
        int bucket = key % BUCKETS;
        return List_Insert(&H->lists[bucket], key);
}

int Hash_Lookup(hash_t *H, int key) {
        int bucket = key % BUCKETS;
        return List_Lookup(&H->lists[bucket], key);
}
```

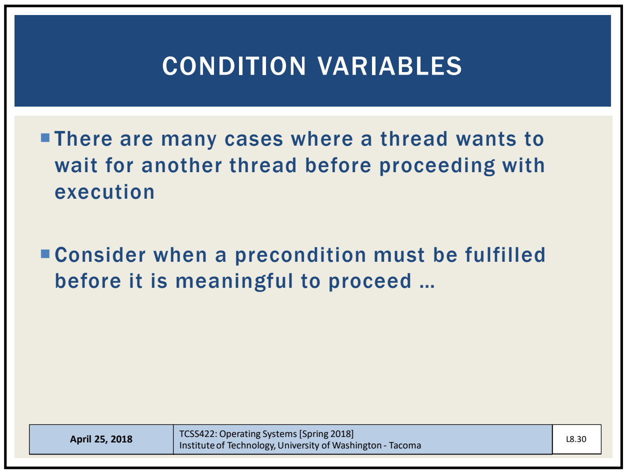


LOCK-FREE DATA STRUCTURES

- Lock-free data structures in Java
- Java.util.concurrent.atomic package
- Classes:
  - AtomicBoolean
  - AtomicInteger
  - AtomicIntegerArray
  - AtomicIntegerFieldUpdater
  - AtomicLong
  - AtomicLongArray
  - AtomicLongFieldUpdater
  - AtomicReference
- See: https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/atomic/package-summary.html

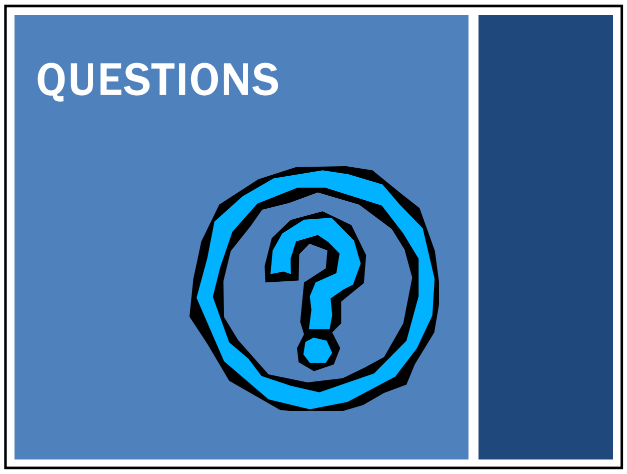


CHAPTER 30 – CONDITION VARIABLES



CONDITION VARIABLES

- There are many cases where a thread wants to wait for another thread before proceeding with execution
- Consider when a precondition must be fulfilled before it is meaningful to proceed …


QUESTIONS